SANDIA REPORT

SAND2013-8361
Unlimited Release
Printed September 2013

Heterogeneous Scalable Framework for Multiphase Flows

Karla Morris

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2013-8361
Unlimited Release
Printed September 2013

Heterogeneous Scalable Framework for Multiphase Flows

Karla Morris,
Reacting Flow Research Department
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551-0969
knmorri@sandia.gov

Abstract

Two categories of challenges confront the developer of computational spray models: those related to the computation and those related to the physics. Regarding the computation, the trend towards heterogeneous, multi- and many-core platforms will require considerable re-engineering of codes written for the current supercomputing platforms. Regarding the physics, accurate methods for transferring mass, momentum and energy from the dispersed phase onto the carrier fluid grid have so far eluded modelers. Significant challenges also lie at the intersection between these two categories. To be competitive, any physics model must be expressible in a parallel algorithm that performs well on evolving computer platforms.

This work created an application based on a software architecture where the physics and software concerns are separated in a way that adds flexibility to both. The develop spray-tracking package includes an application programming interface (API) that abstracts away the platform-dependent parallelization concerns, enabling the scientific programmer to write serial code that the API resolves into parallel processes and threads of execution.

The project also developed the infrastructure required to provide similar API's to other application. The API allow object-oriented Fortran applications direct interaction with Trilinos to support memory management of distributed objects in central processing units (CPU) and graphic processing units (GPU) nodes for applications using C++.

Acknowledgment

This work was funded under LDRD Project Number 158997 and Title "Heterogeneous Scalable Framework for Multiphase Flows".

I would like to offer special thanks to Nicole Lemaster Slattengren, Michael A. Heroux and Maher Salloum for their contributions in the software development of the CTrilinos and ForTrilinos packages. Advice given by Damian Rouson and Joseph Oefelein has been instrumental for the software design aspects of this project. The assistance provided by Sameer Shende and Salvatore Filippone was alse greatly appreciated.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

This work also used hardware resources from the ACISS cluster at the University of Oregon acquired by a Major Research Instrumentation grant from the National Science Foundation, Office of Cyber Infrastructure, "MRI- R2: Acquisition of an Applied Computational Instrument for Scientific Synthesis (ACISS)," Grant #: OCI-0960354.

Contents

List of Figures

Listings

List of Acronyms

GPU Graphic processing unit

CPU Central processing unit

OOP Object-Oriented Programming

STL Standard Templated Library

API Abstract programming interface

MPI Message Passing Interface

TBB Threading Building Blocks

TPL Third party library

PDT Parameterized derived types

TBP Type-bound procedure

PGAS Partitioned Global Address Space

This page intentionally left blank.

Preface

Computational modeling of turbulent combustion is vital for our energy infrastructure and offers the means to develop, test and optimize fuels and engine configurations. In the case of internal combustion engines, fuel injection simulations provide insight into phenomena that determine engine efficiency. In modeling the dilute spray regime, away from the injection site and downstream of the atomization processes, considerable doubts persist regarding how to best parameterize and predict the various couplings between the spray and the surrounding fluid turbulence. These couplings include mass, momentum, and energy exchanges.

The complexity of the relevant physics places considerable demands on computing resources. Hence, the developer of computational spray models confronts challenges related to the computation and those related to the physics. This work developed a spray modeling application to focus on a subset of the computational issues addressed by Fortran 2008 coarrays. Coarrays follow a partitioned global address space (PGAS) model, and provide high-level view of the hardware without reference to the underlying communication layer. Compiler teams are free to implement this feature via any of several open-source or proprietary communication protocols. Coarray codes are therefore able to target multi- and many-core devices with shared and/or distributed memory.

Concurrently with the development of the coarray enable spray modeling application this work developed a novel software architecture that segregates the data computation and communication from the physical models. This architecture will increase the versatility of present and future scientific codes by enabling computational scientists to exploit the countless computing hardware configurations available, and to confront the challenges regarding portability and performance optimization.

The project used the capabilities within C++ Trilinos packages to build the infrastructure to support platform independent, object-oriented Fortran applications. The discrepancies in the features of the programming languages involved (C++, C and Fortran) were overcome through the development of a software infrastructure that emulates generic programming in C and exploits generic and object-oriented programming in Fortran 2003. As a result the work shows new idioms that exploit bleeding-edge features of Fortran and mixed-language programming.

This page intentionally left blank.

Summary

This project developed a novel approach to generic, object-oriented, mixed-language programming. The approach emulates generic programming in C, and exploits actual generic programming features in Fortran 2003. The new concepts, and resulting flexibility are illustrated by constructing a sample application where hardware architecture is encapsulated from the software implementation.

The sample application makes use of the object-oriented Fortran interface to the Trilinos project which is comprised of two layers: CTrilinos and ForTrilinos. CTrilinos ensures software portability by exploiting Fortran 2003 compiler features that provide interoperability with the C programming language. The project involved extensive refactoring of customized scripts first developed by Nicole Lemaster Slattengren. The extended version of the scripts automate part of the glue code generation process by parsing Trilinos C++ source code in the foundational, templated packages Tpetra, Teuchos and Kokkos, and using the results to create the CTrilinos and ForTrilinos glue code.

Tpetra, Teuchos, and Kokkos encapsulate support for platform-independent algorithms in Trilinos. The refactored CTrilinos provides the infrastructure for emulating generic programing in C, which is required to support the Fortran interface, where the generic programing model is based on parameterized derived types (PDTs). ForTrilinos publishes the generic, object-oriented interfaces for direct use in end applications. The CTrilinos package now provides robust, compile-time type checking for ForTrilinos via a unified interface for all instances of a templated class. The significant investment in infrastructure development resulted in a flexible, intuitive object-oriented Fortran interface.

In conjunction with the packages previously discussed this project also developed a spray-tracking application programming interface (API) that abstracts away the platform-dependent parallelization concerns. The particle phase of this application is implemented in Fortran, and makes extensive use of coarrays and object-oriented programming features. Coarrays are a new feature introduced into the Fortran 2008 compiler standard to enable parallel processing using multiple copies of a single program, each copy, is called an image. According to the Partitioned Global Address Space (PGAS) model, each image can access its local data as well as the data from other images though the use of coindices. Scalability of over 85%, in 16834 images, was obtained for the software design followed in the software development of the spray-tracking application.

This page intentionally left blank.

Chapter 1

Introduction

Ongoing research will determine the ultimate configuration for exascale computing hardware. This research has created a moving target for software design. Novel software architectures need to be developed to segregate the data computation and communication from the physical models. These architectures will enable computational scientists to navigate through this transition and will increase the versatility of present and future scientific codes.

The main goal of this work was to create an application based on a software architecture where the physics and software concerns are separated in a way that adds flexibility to both. The developed spray-tracking application programming interface (API) abstracts away the platform-dependent parallelization concerns, and enables the scientific programmer to write serial code that the API resolves into parallel processes and threads of execution. The approach departs from the predominant practice in combustion simulation, wherein the codes that run on leadership-class supercomputers intimately intermesh the physical and computational models. The basic units of data are low-level mathematical constructs, e.g., arrays. Programmers directly manipulate these constructs with a low-level communication mechanism: Message Passing Interface (MPI).

In this project two different approaches were followed to developed the software to support the API. The first approach focused on an API that interacts directly with Trilinos[1], which handles all software concerns for the application. The Trilinos project is an object-oriented software framework with the capabilities required for the solution of large-scale complex multi-physics engineering and scientific problems. At the beginning of this work, Sandia's Trilinos project had the abstract programming interface to support memory management of distributed objects in hardware platforms with central processing units (CPU) and graphic processing units (GPU). However, these capabilities were only accessible to applications using object-oriented C++. The initial stages of the project concentrated on creating support of that same functionality for object-oriented (OO) Fortran applications. A sample application was developed to illustrate the use of Trilinos capabilities that are now available to an OO Fortran application.

The second approach concentrated on an API developed with Fortran coarrays. Coarrays are a partitioned global address space (PGAS) parallel programming model that was incorporated into the Fortran 2008 compiler standard [9]. This new compiler feature provides the language with a simple syntax extension to represent distributed data. At the same time it removes from the programmer all concerns related to how communication takes place in any particular hardware

[1] http://trilinos.sandia.gov

architecture. As part of this work we were able to show excellent scalability results for a partial differential equation solver used a prove of concept for the architecture developed. In the future either API can be linking to existing reactive flow codes, most of which are written in Fortran.

This work developed an object-oriented Fortran interface to template base packages in Trilinos: Tpetra[2] and Kokkos[3]. Tpetra handles the construction and manipulation of distributed objects and provides basic linear algebra functionality, while Kokkos is an abstract programming interface with the capability of managing memory in CPU and GPU nodes. The software abstractions that govern the memory and communication requirements for the tracking of particles in a multiphase flow can be designed to build upon the functionality of these two packages. In this work we developed a software architecture to support a platform independent environment for the particle phase solver using Coarrays. In addition, we have lay the ground work to continue the development of a computational model for two-way coupling of momentum, mass and energy exchanges in multiphase flow in the dilute regime under highly loaded conditions.

In addition to software design, the research also focuses on a critical area of spray modeling. The approach facilitated the complementary development of both software and predictive models. Focusing on sprays allowed testing and optimization of the scientific software; while at the same time provide a platform for future advances in the phenomenological treatment of multiphase flows. Extensive work has been done to date in computational modeling of multiphase reactive and non-reactive flows. Oefelein's [13] simulations of particle-laden flow in a coannular combustor have shown excellent agreement with experimental results for flow conditions similar to those of Sommerfeld [15]. The predictive capabilities of such simulations are limited to unsteady dilute sprays with gas and particle phase under low loading conditions. The coupling between the two phases is then limited to momentum exchange, and the effects of the particles treated as point sources is confined to a single fluid cell. As the Stokes number increases, particles no longer follow fluid streamlines, and the effects of each particles' wake propagates through a larger region in the flow; therefore, the coupling between the gas and particle phase requires accounting for the flow disturbance in various fluid cells.

Interactions with engine manufacturers suggest that providing a better understanding of spray dynamics, even in the non-reacting flow regime, will significantly improve the manufacturers' simulation capabilities. Interactions with computational fluid dynamics software vendors indicate that the current generation of commercial codes has difficulty scaling beyond a few dozen cores. This work developed an application that can be used to address the needs of engine manufacturers by allowing the development of a more accurate and robust computational model for dilute spray simulations. At the same time, the algorithms developed to support the desired software architecture are now available to developers of other applications.

There are technical issues associated with the development of the appropriate software architecture. The design and modeling of the developed software architecture must be able to provide memory management and communication capabilities to any object-oriented Fortran application. This required the creation of an object oriented Fortran interface to Trilinos which is a C++ li-

[2]http://trilinos.sandia.gov/packages/tpetra/

[3]http://trilinos.sandia.gov/packages/kokkos/

brary. The strategy developed to create the interface deals with a variety of interoperability issues, including a unified strategy for matching not only the fundamental C++ data types to their corresponding Fortran intrinsic data types, but also user defined classes in C++ to the corresponding Fortran derived data types.

To achieve a compiler and platform independent interface implementation this project extended the CTrilinos and ForTrilinos packages to include wrappers for Tpetra, Kokkos and additional templated classes within Teuchos[4]. The software design of the wrappers must not only circumvents mismatches in the object-oriented and generic programming capabilities supported by the two languages, but do so in a way that does not put any unnecessary burdens on future ForTrilinos endusers.

The software design of the complete software stack must provide a ForTrilinos interface that satisfies basic portability and usability requirements. Some of the requirements defined during the software development process to address portability and usability concerns are:

- The ForTrilinos interfaces used by object-oriented Fortran applications provide users with a clean syntax to define any specific instantiation of an underlying templated C++ class.
- The ForTrilinos interfaces must provide users with an overloaded single public interface for type-bound procedures (methods in C++) invoked on any instance of an underlying templated C++ class.
- The ForTrilinos object-oriented interface must provide a syntax that feels natural to Fortran developers.
- The ForTrilinos object-oriented interface most provide new idioms for writing classes with generic programming support.
- The object-oriented Fortran wrappers provide compile time error messages in cases were undefined data types are used which provided early error detection to the end user.
- The source code use within each software layer is standard compliant so as to avoid portability problems.

Chapter 2 describes the developed software structure used to construct object-oriented ForTrilinos interfaces for Trilinos packages, Kokkos, Tpetra and Teuchos. The section details the configuration of each software layer and explains how the previously listed requirements are satisfied. A sample object oriented Fortran application is used to illustrate the achieved hardware flexibility.

In developing a multiphase flow model, it is imperative to find a balance between the accuracy of the physics modeled and the demands the model places on the available computational resources. The separation of computation and physics of this project should contribute to a robust application,

[4] http://trilinos.sandia.gov/packages/teuchos/

which will accommodate modifications to the model depending on the desired level of accuracy and the available grid resolution.

Chapter 3 explains the software design of a spray application using Coarray Fortran and chapter 5 presents the conclusions drawn from this work as well as a summary of future work related to this project.

Chapter 2

Hardware Flexibility via Generic Programming

Scientific software applications are developed and expanded through added functionality not only in the pursuit of more accurate and complex multi-physics models but also in an attempt to exploit larger computing platforms of various hardware configurations. High performance computing hardware and algorithm research efforts place scientific software developers in a precarious position as more specific skills are required not only to address scalability and performance in available computing platforms but also the development of more complex multi-physic models.

To efficiently leverage the efforts of scientists, software developed should separate physics and hardware consideration within an application. This separation allows the delegation of each associated requirement to the person with the right set of expertise. In addition, it provides the flexibility to address portability of the scientific application to future hardware configurations.

This chapter discuss the software structure developed to provide the aforementioned flexibility to object-oriented applications. This is accomplished by enabling direct access to capabilities currently available in the Trilinos library.

2.1. Trilinos Library and C++ Template Metaprogramming

The Trilinos project is an object-oriented software framework that provides algorithms and technologies to support the development of large-scale complex multi-physics problems. As such this library encapsulates the implementation of capabilities needed to construct and preform operations on distributed objects commonly used within linear algebra applications. The fundamental packages that comprise the Trilinos library include Teuchos, and the new generation packages Kokkos and Tpetra. These packages abstract from the multi-physics applications all memory management and communication requirements.

Kokkos[1] provides two main capabilities to Trilinos. The first capability is encapsulated in the *Kokkos Node API* which handles memory and specifies work for shared-memory parallel nodes. The second capability is the *Kokkos Linear Algebra Kernel Library*, which contains a collection

[1] http://trilinos.sandia.gov/packages/docs/r10.12/packages/kokkos/doc/html/index.html

of local distributed linear algebra classes and the kernels required for their parallel functionality. The Kokkos Node API includes a series of classes with node definitions that provide support to different hardware configurations:

1. *GPU Nodes use for NVIDIA/CUDA graphic processing units (GPUs).*

   Kokkos::ThrustGPUNode This class provides parallel compute capabilities using the Thrust library.

2. *CPU Nodes use for central processing units (CPUs) standard memory allocation.*

   Kokkos::SerialNode This class provides a simple node with serial execution kernels

   Kokkos::TBBNode This class provides support for multi-core CPUs using the Intel Threading Building Blocks (TBB) library [14].

   Kokkos::TPINode This class provides support for multi-core CPU's using Pthreads TPL.

Kokkos capabilities can be augmented by providing new user defined data structures and kernels for other hardware or library implementations.

Tpetra[2] is a new generation foundational package that has the fundamental data structures and operations required for serial and parallel linear algebra libraries. This package makes extensive use of C++ templates and the Standard Templated Library (STL) to increase functionality by enabling the creation of classes with any defined data type. The classes in this package are templated on a set of parameters of which the most commonly used are:

Scalar: Data type of the data stored within the data structure. The types used are `float`, `double`, `complex<float>` and `complex<double>`

LocalOrdinal: Data type to store the indices of local IDs (`int` in most cases).

GlobalOrdinal: Data type to store the indices of global IDs and global properties of a distributed object. For significantly large distributed objects having different local and global ordinal types could reduce memory requirements. This type can be `long int` or `int`.

Node: Node type defined within Kokkos to enable parallel computation on shared-memory nodes with multi-core CPUs or GPUs.

Teuchos provides Trilinos developers with a set of common tools including BLAS/LAPACK wrappers, smart pointers, parameter lists, etc. In addition, memory management classes in Teuchos, defined reference-counted smart pointers, reference-counted pointers and array classes that are extensively used within Tpetra and other Trilinos packages. These classes replace raw C++ pointers and provide not only functionality for save memory management and usage but also runtime debug checking capabilities [4].

[2] http://trilinos.sandia.gov/packages/docs/r10.12/packages/tpetra/doc/html/index.html

The template metaprogramming technique allows software developers to define classes or functions that are parametrized on a set of template parameters. These generic definitions must be instantiated to generate the appropriate source code that implements the functionality for a specific instance of the templated class or function. Metaprograms defer to compile-time, the actual generation of the instantiated classes as well as the code optimization associate with unrolling of source code [16]. Different studies have correlated software bugs to the presence of duplicated code that its difficult to mantain [3]. Such problems are circumvented in programming languages with full metaprogramming support (i.e. C++ programming language) by the automated source code generation of each required specific implementation.

The generic definition for a template C++ class, `too`, with a template parameter `T`, is shown in listing 2.1. The class has a method (`operations()`) for which the implementation would depend on the data type of the template parameter `T`. The main driver defines an object, `obj`, of class `too<int>`, as a result, the source code of the corresponding implementation for that instantiation is generated at compile-time. The class generic definition, shown in this code example, is complete, even if the main driver included objects of class `too` with other data types for the template parameter.

Listing 2.1: Sample template class too.

```
1  using namespace std;
2
3  template<class T>
4  class too {
5    public:
6     void operations();
7  };
8
9  template <class T>
10 void too<T>::operations()
11 {
12       /* .... */
13 }
14
15 int main()
16 {
17 too<int> obj;
18 obj.operations();
19 }
```

In the Trilinos packages that were briefly described, the template metaprogramming technique enables the development of a library that defines generic distributed classes and operations. The specific implementations are generated at compile-time, base on the instantiated classes within the application using the library. In the case of vectors for instance, the library defines a generic class for distributed vectors, and uses the `Scalar` templated parameter, to allow users access to distributed vectors of any fundamental data type (`double`, `float`, `complex<double>`, etc). The approach contributes to the maintainability of the library's source code, as it avoid manually duplicated code. It also circumvents the need for drastic application refactoring, by supporting applications that can easily exploit classes and functions with a variety of implementations depending on

requirements and available platform configurations.

2.2. Fortran Generic Programming

The Fortran language has generic programming support though not full metaprogramming support. In some cases, the differences between C++ and Fortran render the use of metaprogramming unnecessary in the latter language. Fortran, for example, defines intrinsic data types that are parameterized by a `kind` parameter that defines the actual precision of the data type, as the sample code shown in listing 2.2 [2]. C++, on the other hand, defines several fundamental data types in each data category (Integer data types include int, long, char, short, bool while floating data types are float, double, long double) [8].

Listing 2.2: Use of kind type parameter for double precision integer declaration

```
integer, parameter :: dp = selected_int_kind(9)
integer(kind=dp) :: MaxIndex
```

Fortran generic programing features include parameterized derived types (PDT). This feature builds on the type parameters, of intrinsic and user-defined Fortran data types. The type parameters in intrinsic data types provide the actual precision of the data through a specified `kind` parameter. In user-defined data types the type parameters can be `kind` type parameters or `length` type parameters. A `kind` parameter value most be defined at compile type and it is used to resolved the generic procedure that is referenced. The `length` type parameter can change during execution so it does not have to be defined at compile time [1, 10].

Listing 2.3 shows a proposed Fortran implementation, for the C++ `too<T>` class defined in listing 2.1. The template parameter `T`, takes the form of a `kind` type parameter, in the parameterized derived type, `too(T)`, defined in lines 3-8. The C++ method, `operations()`, is replaced by a type-bound procedure (TBP) with a generic interface of the same name. In lines 10-13, the module shows the implementation for the type-bound procedure `operations_I`, which resolves the overloaded type-bound procedure `operations()` when invoked by an object of type `too` with a `kind_int` type parameter. In the code shown, the main driver does not have access to an `operations` TBP if a different kind parameter is used for an object type `too`.

Listing 2.3: Parameterized data type (PDT) class too.

```
1  module too_module
2    integer , parameter :: kind_int = selected_int_kind(4)
3    type too(T)
4      integer, kind :: T
5     contains
6       procedure :: operations_I
7       generic   :: operations => operations_I
8    end type
9    contains
10      subroutine operations_I(this)
11        class(too(kind_int)), intent(in) :: this
```

```
12        !  .....
13      end subroutine operations_I
14 end module too_module
15 program main
16   use too_module
17   type(too(kind_int)) obj
18   call obj%operations()
19 end
```

As shown in the previous code, PDT and function overloading enable a syntax in object-oriented Fortran that mimics that of template classes in C++. In Fortran, all instantiations and procedure implementations for parameterized derive types must appear explicitly in the source code at compile-time (lines 10-13 in listing 2.3). Due to the lack of compile-time automated instantiation support for generic declarations, an application or library using the generic programming paradigm as supported by PDT could potentially run into maintainability issues. However, issues related to maintainability can be properly mitigated if the automated instantiation is some how emmulated.

2.3. CTrilinos and ForTrilinos

CTrilinos and ForTrilinos are part of the skin packages of Trilinos and provide access to C++ Trilinos functionality from C and Fortran. The original software design of these two packages provided the infrastructure for creating wrappers to core packages such as Epetra, AztecOO, Pliris, Galeri, IFPACK and Amesos. A description of the infrastructure for both CTrilinos and ForTrilinos can be found in previous publications [11, 12]. The original software infrastructure has been modified to support new generation packages and classes that make extensive use of template metaprogramming. This is accomplished by exploiting Fortran 2003 compiler standard features for parameterized data types, and object-oriented programming.

The approach followed to wrap Trilinos, builds on a shadow object interface design, developed by Gray et al, to interface OO C++ with Fortran 95 [6]. In Gray's design, code in a server language, exports a flat interface that grants access to the real object and its functionality. The code in the client language uses a shadow object, which is just a logical interface that allows for the client language to access the real object and its functionality with an object that looks like a native object.

The shadow object approach implemented for Fortrilinos, uses Fortran 2003, which allows us to take advantage of fully object oriented behavior and new C interoperability features. The implementation of this shadow object approach, incorporates two packages, ForTrilinos and CTrilinos. These packages work together to support a full OO Fortran interface to C++ packages within Trilinos. ForTrilinos holds the OO Fortran interfaces that the end user invokes within the Fortran applications. CTrilinos, on the other hand, exist only as a service to ForTrilinos, it enables the portability of the software structure by taking advantages of Fortran's compiler features for interoperability with C. CTrilinos it is not intended as a end user interface.

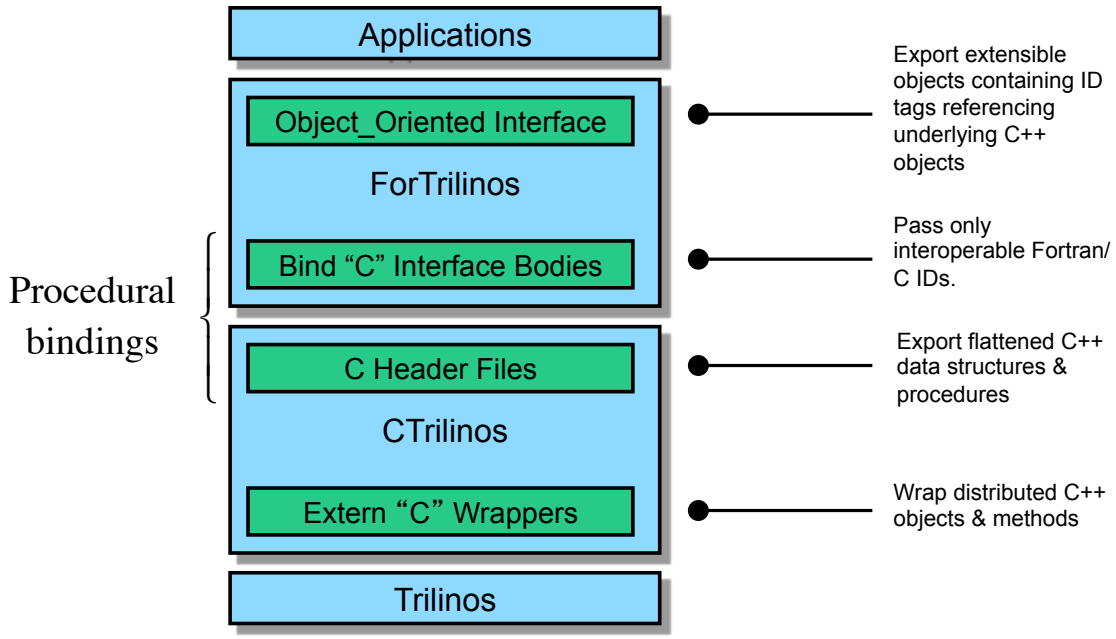

Figure 2.1: Software stack for ForTrilinos enable application.

Figure 2.1 shows a high level representation of the software structure developed to support a ForTrilinos enable application. The OO C++ Trilinos library is at the bottom of the software stack. In this layer we enclose all the C++ classes use to implement the distributed objects and their functionality. In the case of Teuchos and the new generation packages Kokkos and Tpetra, this layer incorporated classes that exploit not only OO class relationships but also classes that use a template metaprogramming paradigm.

The next layer up is the CTrilinos package. This package flattens all C++ data structures and procedures and removes OO features that are not supported by the C programming language. This layer has two sublayers, one with binding C++ code with the extern C attribute, and another with C headers. The CTrilinos layer is there to assure the portability of the software, which is guaranteed by the C interoperability features in the Fortran 2003 standard. This layer circumvents the lack of interoperability support between C++ and Fortran.

The third layer is the ForTrilinos layer, which also has two sublayers. The first sublayer includes the procedural bindings or interface bodies with the `bind(C)` attribute, which correspond to the C headers in the CTrilinos layer. The second Fortrilinos sublayer reintroduces all the OO design features that were removed by the CTrilinos layer. This is the layer that is exposed to the end user.

All binding code is generated automatically by a script that was initially develop by Nicole Lemaster Slattengren. These scripts have undergone extensive modifications to enable the generation of binding code for templated classes and functions. The only overhead associated with the various layers is that of the extra procedural call and table lookups. The real data lives only in the Trilinos layer, ForTrilinos handles only shadow objects. The shadow objects are derived types with a class hierarchy that mirrows the hierarchy of the Trilinos library. Each shadow object has

an ID data member. This ID is a derived data type that holds 3 integers data members with the information required to identify the underlying C++ Trilinos object.

The top layer in the software stack comprises the object-oriented Fortran application codes a user writes by instantiating objects (instances of a Fortran "derived type") and invoking methods ("type-bound procedures" in Fortran nomenclature) on those objects. These objects are lightweight and hold only private identifying information about the underlying C++ objects.

2.4. Multi-Language Software Structure for Generic and Template Meta-Programming Support

This section describes important aspects of the software design used when implementing the previously discussed software stack. The reconciliation of several discrepancies between the programming languages features was not trivial. However, it was necessary to address portability and usability requirements in the object-oriented Fortran interfaces developed to grant Fortran applications access to Trilinos new capabilities.

Design decisions were driven by foreseen needs of end users developing a OO Fortran application. The project studied several possible configuration in order to come up with an approach that provided Fortran developers with similar flexibility already exploited by C++ developers. The required portability of the software stack is satisfied by the use of C interoperability features in the Fortran compiler.

The two main interoperability features used are interoperable kind parameters and the `bind(C)` attribute. Over 30 interoperable kind parameters for several intrinsic Fortran types are provided by the intrinsic module `iso_c_binding` (part of the Fortran 2003 standard). The defined kind parameters insure that the bit representation of a Fortran type matches the corresponding C type provided by the companion C compiler. The `bind(C)` attribute enable the interoperability of derived types and procedures. In the case of procedures a binding label is used to identify the name of the C procedure with the corresponding C function prototype [10].

The flattening of data and procedures in the CTrilinos layer, is required to circumvent the lack of OO and template meta-programming support, and must be managed to insure a scalable approach. Therefore, the code required to support both CTrilinos sublayers and the bottom layer of ForTrilinos (procedural bindings) are created by a customized script. The scripts automate the glue code generation process and enable wrapping templated classes and functions in the C++ library. All possible instantiation for all wrapped template classes are documented in a separate file. The script parses the C++ header files and uses the information to create procedures for each supported instantiation of the template classes.

All C++ objects are referenced in C by identifiable struct IDs. In CTrilinos there is a unique struct ID for each instance of a template class. Listing 2.4 for example, shows the template class `Tpetra::MultiVector`, and the type definition of its template parameters. The parameters in

lines 1-3 are fundamental C++ types and the forth parameter is a Kokkos class. The corresponding CTrilinos type for the previously described instantiation is shown in listing 2.5. The label `_F_I_L_KTPI_` is used to represent each type parameter used in the instantiation (`F=float`, `I=int32_t`, `L=int64_t`, `KTPI=Kokkos::TPINode`). Similar labels are used to differentiate between the procedures that operate on each specific class instantiation.

Listing 2.4: C++ instantiation of templated class

```
1 typedef float Scalar;
2 typedef int32_t LocalOrdinal;
3 typedef int64_t GlobalOrdinal;
4 typedef Kokkos::TPINode Node;
5 typedef Tpetra::MultiVector<Scalar,LocalOrdinal,GlobalOrdinal,Node>Vector;
```

Listing 2.5: CTrilinos C definition of an instant of templated class

```
1 typedef struct {
2   CTrilinos_Table_ID_t table; /*!< Table with reference to the object */
3   int index;          /*!< Array index of the object */
4   boolean is_const; /*!< Whether or not object was declared const */
5 } CT_Tpetra_MultiVector_F_I_L_KTPI_ID_t;
```

Compiler-time checking in the CTrilinos layer, provides a layer of safety to the end user, and it is enable by distinguishing struct IDs for each template class instantiation. To provide support for other instantiations for a template class the CTrilinos or ForTrilinos developer only needs to add the instance to the `template_class` file, used by the script.

The ForTrilinos layer uses struct IDs to reference the underlying C++ objects. In the case of Fortran a single struct ID is use to represent all possible instances of a template class. Listing 2.6 shows the ForTrilinos struct ID for the template class `Tpetra::MultiVector`, where the name used matches that of the CTrilinos layer (minus the template parameters labels). The struct ID is only directly used by Fortran procedure bindings, in the bottom sublayer of ForTrilinos. In this layer both the data structures and procedures have been stripped of any OOP features, so their procedure bindings, which are defined with their corresponding C binding label, keep the necessary information to enable compile-time checking.

Listing 2.6: ForTrilinos Fortran definition of an instant of templated class

```
1   type ,bind(C) :: FT_Epetra_MultiVector_ID_t
2     integer(ForTrilinos_Table_ID_t) :: table
3     integer(c_int) :: index
4     integer(FT_boolean_t) :: is_const
5   end type
```

The OO Fortran syntax that we proposed to support is shown in listing 2.7. Similar to the C++ implementation, template parameters for intrinsic data types are defined in lines 1-3. In this case, the values correspond to interoperable kind parameters that guarantee the appropriate interoperability. The declaration of a parameterized data type must include the value of all kind parameters. The forth parameter is a derived type, but the declaration can not directly use it as a

kind parameter. The lack of support for such functionality can be circumvented, without altering the desired syntax, by using a parameter value, `TPINode_t`, to identify the derived data type.

Listing 2.7: ForTrilinos OO Fortran instantiation of template class

```
1 integer, parameter :: Scalar=c_float
2 integer, parameter :: LocalOrdinal=c_int32_t
3 integer, parameter :: GlobalOrdinal=c_int64_t
4 integer, parameter :: Node=TPINode_t   ! type(TPINode) Node Not supported
5 type(MultiVector(Scalar,LocalOrdinal,GlobalOrdinal,Node))  Vector
```

There are several considerations involved in the implementation of the OO Fortran interface that enables the `MultiVector` PDT declaration. The following section provides a code example that describes the software design of the OO Fortran interfaces and the considerations taken during their implementation.

2.5. Generic and Object-Oriented Programming Paradigms in ForTrilinos

The implementation of the OO Fortran interface of ForTrilinos employes features added to Fortran by the 2003 and 2008 compiler standard. These features include OOP and generic programming provided by PDT. The OOP features include inheritance, operator and function overloading, derived data types, generic interfaces, type-bound procedures, etc. All these features are used without violating the portability requirement of all layers of the software stack.

The PDT declaration shown in the previous section uses interoperable kind parameters to defined the parameterized type parameters. The intrinsic kind parameters defined in the `iso_c_binding` module, previously mentioned, has values that are compiler and platform dependent, and in some cases different parameters have the same value. Due to the lack of full template meta-programming support in Fortran, an explicit procedure implementation is required for each instance of a PDT class. The compiler must be able to differentiate between the different instances and the procedures implemented. This requires unique kind parameters for each instance of a PDT class and unique names for each implementation of the type-bound procedures.

A module implementation for a derived type `foo` is shown in listing 2.8. This PDT has two type parameters (`param_a` and `param_b`). The derived type `foo` is defined in lines 8-23, and each of its available type-bound procedures is defined in lines 11-18. Each type-bound procedure operates on a specific instance of the PDT `foo`. Each instance is defined by the appropriate type parameters values, which are provided by use statement in line 3-5. The name of the subroutine implementing the type-bound procedures has a label with the type parameters of the instance of `foo` that invokes it. The interface for all type-bound procedures is simplified by only publishing a generic interface (lines 19-22). As a result, an application using this PDT module must know only about two methods `First_TBP`, and `Second_TBP`, and not each of the possible four implementations available for each of the instantiations.

The `kind_parameters` module, shown in listing 2.9 is defined to encapsulate all kind parameters, and provides a map representation of unique integer values. The unique values are guaranteed by using the enum construct (lines 5-6). An integer parameter vector is defined with all the interoperable kind parameters supported (lines 7-10) . The enumerated types are use to select the interoperable kind parameter to be use in an intrinsic data type as shown in line 27 of listing 2.8. Although this module could use `c_int` and `c_long` for kind parameters of `int` and `long int` C++ types, the integer data types `c_int32_t` and `c_int64_t` are used instead, for clear differentiation.

Listing 2.8: Module for sample PDT foo.

```
1  module foo_module
2    use iso_c_binding
3    use kind_parameters, only: ft_float_e, ft_double_e, &
4                               ft_int_e, ft_long_e, &
5                               ft_selected
6    private
7    public :: foo
8    type :: foo(param_a,param_b)
9      integer, kind :: param_a, param_b
10   contains
11    procedure :: First_TBP_foo_F_I
12    procedure :: Second_TBP_foo_F_I
13    procedure :: First_TBP_foo_F_L
14    procedure :: Second_TBP_foo_F_L
15    procedure :: First_TBP_foo_D_I
16    procedure :: Second_TBP_foo_D_I
17    procedure :: First_TBP_foo_D_L
18    procedure :: Second_TBP_foo_D_L
19    generic :: First_TBP=>First_TBP_foo_F_I, First_TBP_foo_F_L, &
20                          First_TBP_foo_D_I, First_TBP_foo_D_L
21    generic :: Second_TBP=>Second_TBP_foo_F_I, Second_TBP_foo_F_L, &
22                           Second_TBP_foo_D_I, Second_TBP_foo_D_L
23   end type foo
24 contains
25   subroutine First_TBP_foo_F_I(this,x)
26     class(foo(param_a=ft_float_e,param_b=ft_int_e)), intent(in) :: this
27     real(kind=ft_selected(ft_float_e)) ,intent(in) :: x
28     print *,'call␣to␣First_TBP_foo_F_I␣with␣argument␣x␣kind␣ft_float_e'
29   end subroutine First_TBP_foo_F_I
30   subroutine Second_TBP_foo_F_I(this,y)
31     class(foo(param_a=ft_float_e,param_b=ft_int_e)), intent(in) :: this
32     integer(kind=ft_selected(ft_int_e)) ,intent(in) :: y
33     print *,'call␣to␣Second_TBP_foo_F_I␣with␣argument␣y␣kind␣ft_int_e'
34   end subroutine Second_TBP_foo_F_I
35   subroutine First_TBP_foo_F_L(this,x)
36     class(foo(param_a=ft_float_e,param_b=ft_long_e)), intent(in) :: this
37     real(kind=ft_selected(ft_float_e)) ,intent(in) :: x
38     print *,'call␣to␣First_TBP_foo_F_L␣with␣argument␣x␣kind␣ft_float_e'
39   end subroutine First_TBP_foo_F_L
40   subroutine Second_TBP_foo_F_L(this,y)
41     class(foo(param_a=ft_float_e,param_b=ft_long_e)), intent(in) :: this
42     integer(kind=ft_selected(ft_long_e)) ,intent(in) :: y
43     print *,'call␣to␣Second_TBP_foo_F_L␣with␣argument␣y␣kind␣ft_long_e'
```

```
44    end subroutine Second_TBP_foo_F_L
45    subroutine First_TBP_foo_D_I(this,x)
46      class(foo(param_a=ft_double_e,param_b=ft_int_e)), intent(in) :: this
47      real(kind=ft_selected(ft_double_e)) ,intent(in) :: x
48      print *,'call␣to␣First_TBP_foo_D_I␣with␣argument␣x␣kind␣ft_double_e'
49    end subroutine First_TBP_foo_D_I
50    subroutine Second_TBP_foo_D_I(this,y)
51      class(foo(param_a=ft_double_e,param_b=ft_int_e)), intent(in) :: this
52      integer(kind=ft_selected(ft_int_e)) ,intent(in) :: y
53      print *,'call␣to␣Second_TBP_foo_D_I␣with␣argument␣y␣kind␣ft_int_e'
54    end subroutine Second_TBP_foo_D_I
55    subroutine First_TBP_foo_D_L(this,x)
56      class(foo(param_a=ft_double_e,param_b=ft_long_e)), intent(in) :: this
57      real(kind=ft_selected(ft_double_e)) ,intent(in) :: x
58      print *,'call␣to␣First_TBP_foo_D_L␣with␣argument␣x␣kind␣ft_double_e'
59    end subroutine First_TBP_foo_D_L
60    subroutine Second_TBP_foo_D_L(this,y)
61      class(foo(param_a=ft_double_e,param_b=ft_long_e)), intent(in) :: this
62      integer(kind=ft_selected(ft_long_e)) ,intent(in) :: y
63      print *,'call␣to␣Second_TBP_foo_D_L␣with␣argument␣y␣kind␣ft_long_e'
64    end subroutine Second_TBP_foo_D_L
65  end module foo_module
```

Listing 2.9: Module with kind type parameters definitions for PDT foo.

```
1  module kind_parameters
2   use iso_c_binding, only: c_int, c_int32_t, c_int64_t, c_float, c_double
3   integer(c_int) ,parameter :: ft_kind_e = c_int
4   enum, bind(c)
5    enumerator :: ft_int_e=1, ft_long_e, ft_float_e, ft_double_e
6   end enum
7   integer,parameter,dimension(4) ::ft_selected=(/c_int32_t,&
8                                                  c_int64_t,&
9                                                  c_float,&
10                                                 c_double/)
11 end module kind_parameters
```

An external application using the PDT `foo` is shown in listing 2.10. All type parameters required for the instantiation of a PDT `foo` object are encapsulated in `my_types` module in lines 1-7. This provides flexibility to the application since any changes need to access a different instantiation and its corresponding functionality is limited to that module. Main declared two different instances of a `foo` object `foo_FI_inst` and `foo_DL_ints`. For consistency and again to increase the flexibility of the overall application helper variables of intrinsic data types are declared using the kind parameters defined in the PDT instantiation. Lines 18 to 21 show the invocation of the two type-bound procedures by each instance of `foo`. A sample output is included in listing 2.11 verifying that the appropriate TBP implementation was called. The implementation shown provides compile-time errors in the case of type mismatch.

Listing 2.10: Sample main making use of PDT foo.

```
1  module my_types
2    use kind_parameters
```

```
3   integer, parameter :: D=ft_double_e
4   integer, parameter :: L=ft_long_e
5   integer, parameter :: F=ft_float_e
6   integer, parameter :: I=ft_int_e
7 end module
8 program main
9   use my_types
10  use foo_module
11  type(foo(F,I)) :: foo_FI_inst
12  type(foo(D,L)) :: foo_DL_inst
13  real(kind=ft_selected(F)) :: value_F=10.0
14  integer(kind=ft_selected(I)) :: index_I=1
15  real(kind=ft_selected(D)) :: value_D=200.0
16  integer(kind=ft_selected(L)) :: index_L=20
17
18  call foo_FI_inst%First_TBP(value_F)
19  call foo_FI_inst%Second_TBP(index_I)
20  call foo_DL_inst%First_TBP(value_D)
21  call foo_DL_inst%Second_TBP(index_L)
22 end program main
```

Listing 2.11: Output from running example application using the PDT foo.

```
1 call to First_TBP_foo_F_I with argument x kind ft_float_e
2 call to Second_TBP_foo_F_I with argument y kind ft_int_e
3 call to First_TBP_foo_D_L with argument x kind ft_double_e
4 call to Second_TBP_foo_D_L with argument y kind ft_long_e
```

This PDT design supports the desired syntax, but it requires extreme source code duplication. In the absent of full template metaprogramming support, a script has been developed to automate source code generation based on a simple interface or skeleton, such as the one shown in listing 2.12. The scripts follow a similar approach to that of the Forpedo[3] project. The project was developed to emulate run time polymorphism in earlier versions of the Fortran compiler. The For-Trilinos project adopt a similar skeleton syntax and developed a script to expand the skeleton of the PDT class by providing the implementation for all procedure instantiations. The skeleton requires the definition of a string to be replaced, the label of the type parameter, the value of the type parameter and the declaration for an intrinsic data type with that kind parameter (see lines 1-4).

Listing 2.12: Module skeleton for sample PDT foo.

```
1 #definetype FirstType  F ft_float_e ft_selected(ft_float_e)
2 #definetype FirstType D ft_double_e ft_selected(ft_double_e)
3 #definetype SecondType I ft_int_e ft_selected(ft_int_e)
4 #definetype SecondType L ft_long_e ft_selected(ft_long_e)
5 module foo_module
6   use iso_c_binding
7   use kind_parameters, only: ft_float_e, ft_double_e, &
8                              ft_int_e, ft_long_e, &
9                              ft_selected
10  private
```

[3] http://fortranwiki.org/fortran/show/Forpedo

```
  public :: foo
  type :: foo(param_a,param_b)
    integer, kind :: param_a, param_b
  contains
   #procedure_start
   procedure :: First_TBP_foo_<FirstType>_<SecondType>
   procedure :: Second_TBP_foo_<FirstType>_<SecondType>
   #procedure_end
   #generic_start
   generic :: First_TBP=>First_TBP_foo_<FirstType>_<SecondType>
   generic :: Second_TBP=>Second_TBP_foo_<FirstType>_<SecondType>
   #generic_end
  end type foo
contains
  #procedure_impl_start
  subroutine First_TBP_foo_<FirstType>_<SecondType>(this,x)
    class(foo(param_a=~FirstType,param_b=~SecondType)), intent(in) :: this
    real(kind=@FirstType) ,intent(in) :: x
    print *,'call to First_TBP_foo_<FirstType>_<SecondType> with argument
       x kind ~FirstType'
  end subroutine First_TBP_foo_<FirstType>_<SecondType>
  subroutine Second_TBP_foo_<FirstType>_<SecondType>(this,y)
    class(foo(param_a=~FirstType,param_b=~SecondType)), intent(in) :: this
    integer(kind=@SecondType) ,intent(in) :: y
    print *,'call to Second_TBP_foo_<FirstType>_<SecondType> with argument
       y kind ~SecondType'
  end subroutine Second_TBP_foo_<FirstType>_<SecondType>
  #procedure_impl_end
#conclusion_start
end module foo_module
#conclusion_end
```

This page intentionally left blank.

Chapter 3

Platform-agnostic multiphase flow application via Fortran 2008 coarrays

Fortran has always been a language with a focus on high efficiency for numerical computations on array data sets. Over the past 10-15 years, it has picked up features from mainstream programming, such as class abstractions, but also catered to its prime users by developing a rich set of high-level array operations. Controlling the flow of information allows for a purely functional style of expressions that is expressions that rely solely upon functions that have no side effects. Side effects influence the global state of the computer beyond the function's local variables. Examples of side effects include input/output, modifying arguments, halting execution, modifying non-local data, and synchronizing parallel processes. There have been longstanding calls for employing functional programming as part of the solution to programming parallel computers [5]. The Fortran 2008 standard also includes a parallel programming model based primarily upon the coarray distributed data structure. The advent of support for Fortran 2008 coarrays in the Cray and Intel compilers makes the time ripe to explore synergies between Fortran's explicit support for functional expressions and coarray parallel programming [7].

A sample main driver that uses the spay tracking application programming interface developed is shown in listing 3.1. The application is fully distributed even though no explicit library calls are made. The platform-agnostic multiphase flow application is implemented via coarrays. Coarrays (new feature introduced into the Fortran 2008 compiler standard) enable parallel processing using multiple copies of a single program, each copy, is called an image. According to the Partitioned Global Address Space (PGAS) model, each image can access its local data as well as the data from other images though the use of coindices. In this programming model all communication is shown explicitly by the use of a coindices.

Listing 3.1: Source code for object-oriented Fortran Multiphase flow application driver.

```
1  program main
2    use ForTrilinos_assertion_utility, only : assert,error_message
3    use math_utility, only : error_within_tolerance
4    use kind_parameters, only : rkind, ikind
5    use math_constants, only : zero, local_pmax, a, b, c, half
6    use cartesian_grid_implementation, only: c_grid
7    use fluid_implementation, only: carrier_fluid
8    use local_particles_implementation, only : local_particles
9    use particles_implementation, only: particles
10   implicit none
```

```
!> @name Time Advanced Particle Test
!! @{

!> <BR> Runge Kutta 4th order quadrature algorithm
!> @brief Time Advanced Particle Phase
!> Tracks particle positon and velocity as a function of time using RK4.

! Variable declaration
type(c_grid)                :: grid
type(carrier_fluid)         :: gas
type(particles), save       :: spray, k1, k2, k3, spray_temp
real(rkind)           :: time=0.0_rkind, dt=0.0165_rkind, t_final=20
    _rkind
integer(ikind)        :: i, istep=0, istep_final=100
character(len=*), parameter :: mesh_filename_root='CG3D_'
character(len=100)  :: mesh_filename
character(len=*), parameter :: fluid_filename_root='TG2DF_CG3D_'
character(len=100)  :: fluid_filename
character(len=*), parameter :: spray_filename_root='RandomSpray_CG3D_'
character(len=100)  :: spray_filename
character(len=*), parameter :: connectivity_filename='TG2DF_CG3D_connect
    .txt'

! Reading intialization data
write(mesh_filename,'(A,I6.6,A)') mesh_filename_root,this_image(),'.tec'
call grid%new_c_grid(mesh_filename,connectivity_filename)
write(fluid_filename,'(A,I6.6,A)') fluid_filename_root,this_image(),'.
    tec'
call gas%new_carrier_fluid(grid,fluid_filename)
write(spray_filename,'(A,I6.6,A)') spray_filename_root,this_image(),'.
    tec'

! Contructing distributed data objects
call spray%new_particles(gas,spray_filename,time)
call spray_temp%new_particles()
call k1%new_particles()
call k2%new_particles()
call k3%new_particles()

! Time advancing multiphase flow
do while (time<t_final)
  call spray%interpolate()

  ! Implementation of RK4
  k1 = spray%t()*dt
  spray_temp = spray + k1*half
  k2 = spray_temp%t()*dt
  spray_temp = spray + k2*half
  k3 = spray_temp%t()*dt
  spray_temp = spray + k3*c
  spray = spray + k1*a + k2*b + k3*b + spray_temp%t()*(dt*a)

  time = spray%get_time()
  if (this_image()==1) print * , 'TIME=', istep, time
```

```
61
62      ! Distributed object data relocation
63      call spray%all_images_relocate_particles()
64      sync all
65
66      ! Output file for visualization
67      istep = istep + 1
68      if(mod(istep,5)==0) call spray%output('RandomSpray_CG3D_')
69    enddo
70
71    call spray%output('RandomSpray_Final_')
72  end
```

The functional and object-oriented programing approaches used in the implementation of this spray modeling application contribute to an expressive syntax. The distributed operators implemented (1^{st} derivative with respect to time %t, addition +, multiplication -, etc.) are able to support the Runge Kutta fourth order method used within the main application driver to time advance the spray phase (see lines 51-57 in listing 3.1). In addition, these operators can also be used to support other time advancing schemes with the same expressive syntax. The unified modeling language class diagram, shown in figure 3.1, provides a high level description of the software design of the overall software infrastructure. The project has published excellent scalability results (87% weak scaling for up to 16384 images) for a prototype applications using the a similar software design [7].

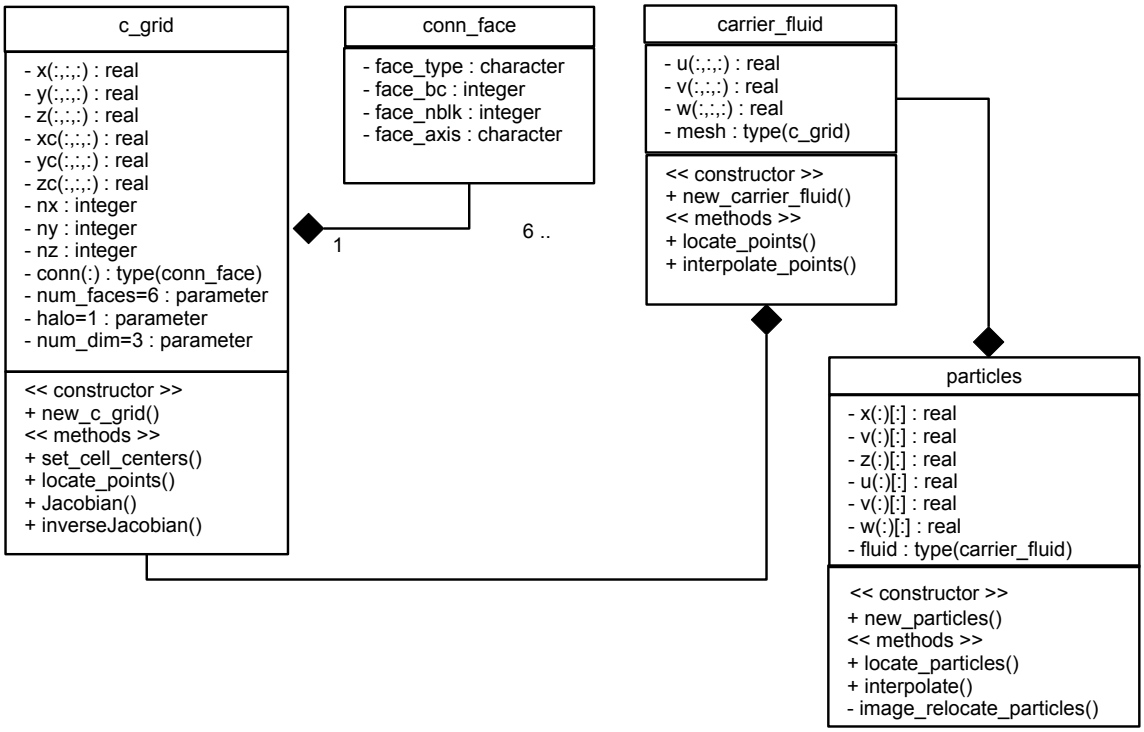

Figure 3.1: Unified modeling language (UML) class diagram for MPFlows software package.

This page intentionally left blank.

Chapter 4

Results

This report describes the software design used to allow object-oriented Fortran applications direct access to capabilities within the Trilinos C++ library. Although the outlined approached is compiler standard compliant, the extensive use of very sophisticated features, not commonly used by software developers, contributes to the bleeding edge nature of the work and limited compiler support. The current implementation of CTrilinos and ForTrilinos wrappers for Tpetra, Kokkos and Teuchos has been tested with the IBM XL compiler, version 13.1. The CTrilinos package and the ForTrilinos procedural binding, for the aforementioned wrapped packages, work on GNU 4.9. Due to the lack of PDT support in the GNU Fortran compiler the object-oriented Fortran interfaces in ForTrilinos are not available for Tpetra, and Kokkos.

The software design approach used throughout this project was mainly driven by an effort to address portability and usability requirements. These requirements and the aspects of the software design implemented to satisfy them, can be summarized as follow.

- *The ForTrilinos interfaces used by object-oriented Fortran applications must provide users with a clean syntax to define any specific instantiation of an underlying C++ template class.*

  The software design of the OO Fortran interfaces uses a PDT for each shadow class used to access the underlying C++ template class. Each of the `kind` type parameters in the PDT shadow class corresponds to a template parameter in the template C++ class.

- *The ForTrilinos interfaces must provide users with an overloaded single public interface for type-bound procedures (methods in C++) invoked on any instance of an underlying template C++ class.*

  For each C++ generic method in a template class, the PDT shadow class publishes a corresponding overloaded interface. Within the module implementing the PDT shadow class there is a type-bound procedure specified for each instantiation or combination of valid kind type parameters defined in the PDT shadow class. Users access the functionality only through the overloaded interface, which simplifies the interface of the end user application.

- *The ForTrilinos object-oriented interface must provide a syntax that feels natural to Fortran developers.*

  There are several issues to address in order to achieve a natural Fortran syntax, without

compromising the portability of the software stack. Since C does not support the C++ bool type, CTrilinos establishes a custom, integer Boolean type for use with C compilers and uses this type to represent C++ bool values. With this convention, the following CTrilinos code defines the employed Boolean true and false shown below.

```
typedef int boolean;
#ifndef TRUE
# define TRUE 1
#endif
#ifndef FALSE
# define FALSE 0
#endif
```

Respecting this convention, ForTrilinos avoids the interoperable Fortran 2003 `c_bool` kind parameter and instead employs a corresponding `FT_boolean_t` integer kind parameter within the procedural bindings. The implementation of the OO Fortran interfaces in ForTrilinos make the necessary conversions so all published interfaces use the intrinsic Fortran data type `logical`.

To achieve a natural Fortran syntax `Teuchos::Array` classes implemented in Trilinos are not directly used within the OO ForTrilinos interface. A separate `ForTrilinos_PDT_utils.F90` module is implemented to handle all conversions between `Teuchos::Array` classes and Fortran allocatable arrays. The use of the module functionality within the implementation of OO Fortran interfaces once again make all conversions and the end user passes and receives data using only the allocatable arrays.

- *The ForTrilinos object-oriented interface most provide new idioms for writing classes with generic programming support.*

  The PDT feature provide for generic programming support in Fortran have been used in this project to develop Fortran interfaces for C++ template classes. The portability problems that arise due to kind parameters, compiler and platform dependencies, are circumvented through the use of enumerated types which provide unique values for each of the PDT type parameters. The script developed as part of this project to automate the generation of source code for each PDT instantiation enables a generic programming support that mimics more closely that of the template metaprograming paradigm supported within the C++ library.

- *The object-oriented Fortran wrappers provide compile time error messages in cases were undefined data types are used which provides early error detection to the end user.*

  Source code errors can be better addressed if early detection is possible. The were several software designs considered to implement the object-oriented shadow objects. Some designs were discarded due to their deferred error detection behavior, where the presence of an unsupported instance of a PDT, type-bound procedure or implementation for an overloaded procedure is discovered only at run-time. The current design implemented in ForTrilinos circumvents this behavior and provides users with compile-time error checking features for all unsupported instances of PDT and procedures.

- *The source code use within each software layer is standard compliant so as to avoid portability problems.*

  The software design implemented in each layer is compiler standard compliant. Special compiler vendor features are avoided to prevent portability problems. The features used provide an elegant solution for the multi-programming language environment, however, they have not been extensively used by developers. Compiler vendor support would increase by out interaction with different compiler teams.

This page intentionally left blank.

Chapter 5

Conclusion and Future Work

Computational modeling of turbulent combustion is vital for our energy infrastructure and offer the means to develop, test and optimize fuels and engine configurations. In the case of internal combustion engines, fuel injection simulations provide insight into the required calibration for appropriate turbulent mixing and efficient combustion. In modeling the dilute spray regime, away from the injection site and downstream of the atomization processes, considerable doubts persist regarding how to best parameterize and predict the various two-way couplings between the dispersed phase and the surrounding fluid turbulence. These couplings include mass, momentum, and energy exchanges.

Two categories of challenges confront the developer of computational spray models: those related to the computation and those related to the physics. Regarding the computation, the trend towards heterogeneous, multi- and many-core platforms will require considerable re-engineering of codes written for the current supercomputing platforms. Regarding the physics, accurate methods for transferring mass, momentum and energy from the dispersed phase onto the carrier fluid grid have so far eluded modelers. Significant challenges also lie at the intersection between these two categories. To be competitive, any physics model must be expressible in a parallel algorithm that performs well on evolving computer platforms.

This project laid the foundation to tackled these two challenges in scientific software applications developed with the Fortran programming language. Fortran is the predominant language of choice within the combustion community. The computational challenges have been addressed by designing a software infrastructure to allow Fortran application direct access to C++ Trilinos capabilities that encapsulate hardware communication and computation dependencies. Computational challenges have also been addressed through the concurrent development of a spray modeling application using the parallel programming model included in the Fortran compiler standard.

Additional work is required to improve the accuracy of the physical models. The current spray model application was developed with a set of distributed data structures, but the parametric functions that model the attenuation of fluid turbulent properties due to the presence of the dispersed particles have not been incorporated. The software design of the application encapsulated the properties of the particle phase in data structures that don't need to be modified as the parametric functions are implemented. The particle phase flow application must be tested when interfaced with already available fluid phase flow software. The previously mentioned parametric functions will extend the capabilities of the particle phase to account for deformation and wake effects on the background fluid affected by large scale particles.

The platform-agnostic multiphase flow application developed makes use of several programming paradigms; object-oriented, functional programming and parallel programming. All of which are currently supported as part of the Fortran 2008 compiler standard. The combination of functional programming and the implementation of data type calculus design pattern provide an expressive syntax to the application and use objects that support distributed data without third party library dependencies. This worked was able to obtained great scalability results for a prototype application developed with the same software design implemented in the spray modeling software application.

Two sets of scripts were developed in the course of this project. One of the scripts was refactor from Nicole Lemasters Slattengren original CTrilinos and ForTrilinos customized script. The added functionality of the script enable parsing Trilinos C++ source code in the foundational, template packages Tpetra, Teuchos and Kokkos, and use the results to create the CTrilinos and ForTrilinos glue code required to access their provided capabilities. The second script developed automates the generation of source code for each specific instantiation of a PDT. It does so based on a provided skeleton file, which can be thought of as a generic interface for all of the PDT and the corresponding type-bound procedures.

The CTrilinos and ForTrilinos wrappers for Tpetra, Kokkos and Teuchos are available to any Trilinos library developer, as well as software developers within Sandia. The wrappers have not been publicly released. The appendix section of this report contains the source code for an end users application implemented using the OO Fortran interfaces. In the future close interaction with compiler vendors is required to increase support for all the compiler features use within the software stack that was developed in this project. This sort of interaction is responsible for the increase in ForTrilinos compiler support we have experienced in the last couple of years, which has gone from one compiler vendor supporting the first release to four compilers in the last release.

Taking this work to the next level requires merging the wrappers implementations of CTrilinos and ForTrilinos into the release branch of Trilinos to make the capabilities available to non Sandia scientist. The extensive use of both packages would inadvertently contribute to an increase in compiler vendor support.

References

[1] Jeanne C. Adams, Walter S. Brainerd, Richard A. Hendrickson, Richard E. Maine, Jeanne T. Martin, and Brian T. Smith. *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*. Springer, 2009.

[2] E. Akin. *Object-Oriented Programming via Fortran 90/95*. Cambridge University Press, 2003.

[3] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy. Clone smells in software evolution. In *Software Maintenance. ICSM 2007. IEEE International Conference*, October 2-5 2007.

[4] Roscoe Barlett. Teuchos C++ memory management classes, idioms, and related topics the complete reference a comprehensive strategy for safe and efficient memory management in C++ for high performance computing. SANDIA Report 2010-2234, Sandia National Laboratory., 2011.

[5] D. C. Can. Retire Fortra? adebate rekindle. *Commun. ACM*, 35.

[6] M. G. Gray, R. M. Roberts, and T. M. Evans. Shadow-object interface between Fortran 95 and C++. *Computing in Science and Engineering*, 1(2):63–70, 1999.

[7] Magne Haveraaen, Karla Morris, and Damian Rouson. High-performance design patterns for modern fortran. In *First International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, November 22 2013.

[8] Debasish Jana. *C++ and Object-Oriented Programming Paradigm*. Prentice-Hall, 2005.

[9] ISO/IEC JTC1/SC22/WG5/N1830. Information technology - programming languages - fortran - part 1: Base language. ISO/IEC DIS 1539-1, International Organization for Standardization/International Electrotechnical Commission, 2010.

[10] M. Metcalf, J. K. Reid, and M. Cohen. *Modern Fortran Explained*. Oxford University Press, 2011.

[11] K. Morris, D. W. I. Rouson, and M. N. Lemaster. On the scalable development of portable object-oriented fortran interfaces to C++: A PDE solver prototype. *ACM Trans. Math Soft.*, in review 2013.

[12] K. Morris, D. W. I. Rouson, M. N. Lemaster, and Salvatore Filippone. Exploring capabilities within fortrilinos by solving the 3D burgers equation. *Scientific Programming*, 20(3):275–292, 2012.

[13] Joseph C. Oefelein. Large eddy simulation of turbulent combustion processes in propulsion and power systems. *Progress in Aerospace Science*, 42:2–37, 2006.

[14] James Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[15] M. Sommerfeld and H. Qiu. Characterization of particle-laden, confined swirling flows by phase-doppler anemometry and numerical calculation. *International Journal of Multiphase Flow*, 19(6):1093–1127, 1993.

[16] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4), 1995.

Appendix A

Source Code For Sample Application

This section provides the complete source code for a sample application that can use any of the Kokkos defined nodes. Each example access underlying C++ Trilinos functionality through the available wrappers in a specific layer of the software stack shown in figure 2.1.

A.1. Trilinos Sample Application

A prototype application for a distributed, scalable and portable vector matrix multiplication using Trilinos is shown in listing A.1. This prototype application can run on any platform for which a Kokkos node has been implemented. The application source code does not need to be refactor for portability since this information is encapsulated in the `Node` type defined in line 27. All parameters required by templated classes are defined in lines 22-28. The parameters are used to instantiate the template classes that implement distributed sparse matrix and vector objects in lines 29 and 30 respectively. The library informs the application of the platform and build configuration by the `Node` definition return by the `Tpetra::DefaultPlatfrom` class (line 40). The communicator information and functionality is contained within the `Teuchos::Comm` class (line 41) , which provides wrappers to data communication procedure such as `broadcast`, `reduceAll`, `gatherAll`, etc.

The data of all objects is distributed base on a map. The `Tpetra:Map` class holds local/global indices and properties information required for communication and operations on the data. The map object is created in line 45-48 by invoking one of the overloaded constructors. Local array functionality is provided in the Trilinos library by the `Teuchos` package. The different array classes within Teuchos wrap C++ raw pointers with reference counted pointers to manage dynamically allocated memory. These arrays serve as light weight replacements for raw pointers and are pass and return as arguments to functions. Lines 50 and 51 declare an Array with constant values of type `GlobalOrdinal` and populates the array elements with the return value form then `getNodeElementList` method respectively. The sparse matrix object is constructed in line 55, and data is inserted one row at the time (lines 57-75) by using the Tuple class, a compile-time array that allow the array argument to be constructed and passed to the function on the fly. A vector matrix multiplication method is invoked in line 92, using the previously constructed distributed vectors (lines 81-82). Even in the absence of any explicit parallel library call the source code executes distributed operations.

Listing A.1: Source code for sample C++ application using Trilinos library.

```
#include "Teuchos_GlobalMPISession.hpp"
#include "Teuchos_oblackholestream.hpp"
#include "Teuchos_Array.hpp"
#include "Tpetra_DefaultPlatform.hpp"

#include "Kokkos_DefaultKernels.hpp"
#include "Kokkos_DefaultNode.hpp"
#include "Kokkos_DefaultSparseOps.hpp"

#include "Tpetra_Map.hpp"
#include "Tpetra_MultiVector.hpp"
#include "Tpetra_CrsMatrix.hpp"

#include "iostream"

int main( int argc, char *argv[])
{
  Teuchos::oblackholestream blackhole;
  Teuchos::GlobalMPISession mpiSession(&argc,&argv,&blackhole);

  //Specify types use in this example
  typedef double                                          Scalar;
  typedef int                                              LocalOrdinal;
  typedef int                                              Ordinal;
  typedef int                                              GlobalOrdinal;
  typedef Tpetra::DefaultPlatform::DefaultPlatformType  Platform;
  typedef Tpetra::DefaultPlatform::DefaultPlatformType::NodeType  Node;
  typedef Kokkos::DefaultKernels<Scalar,LocalOrdinal,Node>::SparseOps DSM;
  typedef Tpetra::CrsMatrix<Scalar,GlobalOrdinal,GlobalOrdinal,Node,DSM>
     CrsMatrix;
  typedef Tpetra::Vector<Scalar,LocalOrdinal,GlobalOrdinal,Node> Vector;
  using Teuchos::RCP;
  using Teuchos::tuple;

  //Parameter
  int numGlobalElements = 40;
  int numVec =1;
  Scalar alpha=1.0,beta=0.0;

  // Get communicator
  Platform &platform = Tpetra::DefaultPlatform::getDefaultPlatform();
  RCP<const Teuchos::Comm<Ordinal> > comm = platform.getComm();
  RCP<Node>                          node = platform.getNode();

  // Create map
  RCP<const Tpetra::Map<LocalOrdinal,GlobalOrdinal,Node> > map;
  map =
  Tpetra::createUniformContigMapWithNode<LocalOrdinal,GlobalOrdinal,Node>
     (numGlobalElements, comm, node);
  const size_t numMyElements = map->getNodeNumElements();
  Teuchos::ArrayView<const GlobalOrdinal> myGlobalElements;
  myGlobalElements = map->getNodeElementList();
```

```
52
53    // Create a CrsMatrix using the map, with a dynamic
54    // allocation of 3 entries per row
55    RCP<CrsMatrix> A = rcp ( new CrsMatrix(map,3));
56    // Add rows one-at-a-time
57    for (size_t i=0; i<numMyElements; i++) {
58      if (myGlobalElements[i] == 0) {
59        A->insertGlobalValues(myGlobalElements[i],
60           tuple<GlobalOrdinal>(myGlobalElements[i],myGlobalElements[i]+1),
61           tuple<Scalar> (2.0,-1.0));
62
63      }
64      else if (myGlobalElements[i] == numGlobalElements-1) {
65        A->insertGlobalValues(myGlobalElements[i],
66            tuple<GlobalOrdinal>(myGlobalElements[i]-1,myGlobalElements[i]),
67            tuple<Scalar> (-1.0,2.0));
68      }
69      else {
70        A->insertGlobalValues(myGlobalElements[i],
71           tuple<GlobalOrdinal>(myGlobalElements[i]-1,
72                                myGlobalElements[i],myGlobalElements[i]+1),
73           tuple<Scalar> (-1.0,2.0,-1.0));
74      }
75    }
76
77    // Complete the fill, ask that storage be reallocated and optimized
78    A->fillComplete();
79
80    // Create MultiVectors
81    RCP<Vector> X = rcp (new Vector (map));
82    RCP<Vector> Y = rcp (new Vector (map));
83
84    // Insert values on MultiVector X
85    for (size_t i=0; i<numMyElements; i++) {
86       const Scalar value=1.0*myGlobalElements[i];
87       X->replaceGlobalValue(myGlobalElements[i],value);
88    }
89
90
91    // Matrix-Vector Multiply
92    A-> apply(*X,*Y);
93
94   //ÂãOutput Y
95   Teuchos::ArrayRCP<Scalar> Yval;
96   Yval=Y->getDataNonConst(0);
97   for (size_t i=0; i<numMyElements; i++)
98   {
99     std::cout << Yval[i] <<"␣␣" << myGlobalElements[i]   << std::endl;
100   }
101
102 }
```

A.2. CTrilinos Sample Application

CTrilinos is not intended as a user interface, but for testing purposes an equivalent implementation of the application shown in listing A.1 was developed using the interfaces defined in the CTrilinos software layer (see listing A.2). The lack of OOP support in the C language makes the procedures and arguments in this layer a lot more complex when compared to their C++ counterparts. No inheritance and function overloading support forces this layer to create unique names for each function implementation. Each instantiation of the templated classes and functions must also have a unique name.

All classes in C++ are identified by a struct id as shown in line 50. The variables that hold the structs used to identified the underlying C++ objects within the application are declared in lines 48-70. The struct names have the package and class name followed by a label for the specific data type of the templated parameters. For example, in line 60, the struct id of type `CT_Teuchos_ArrayRCP_D_ID_t` corresponds to an object of type `Teuchos::ArrayRCP<double>`.

The objects that encapsulate the functionality related to platform configuration and the instantiation of the `comm` and `node` objects use for managing data distribution, communication and operations are created in lines 83-85. The `Map` and `MultiVector` objects are created in lines 88 and 100 respectively. In this implementation the local array objects that are passed as arguments to the function `Tpetra_CrsMatrix_D_I_I_KTPI_KDS_insertGlobalValues` (`insertGlobalValues` method in C++) must be created separately before they are used (see lines 107-113 and 123-129). The matrix vector multiplication procedure is invoked in line 167. Before exiting the application all objects are destroyed in lines 176-186.

Listing A.2: Source code for sample C application using CTrilinos library package.

```
1  #include "assert.h"
2  #include "stdlib.h"
3  #include "stdio.h"
4
5  #include "mpi.h"
6
7  #ifdef HAVE_MALLOC_H
8  #ifndef __APPLE__
9  #include "malloc.h"
10 #else
11 #include "sys/malloc.h"
12 #endif
13 #endif
14
15 #include "CTrilinos_config.h"
16 #include "CTrilinos_enums.h"
17 #include "CTrilinos_flex_enums.h"
18 #include "CTrilinos_table_man.h"
19
20 #include "CTeuchos_Comm.h"
21 #include "CTeuchos_ArrayView.h"
22 #include "CTeuchos_ArrayRCP.h"
23 #include "CTeuchos_Tuple.h"
```

```
24 #include "CTeuchos_Array.h"
25 #include "CTeuchos_ParameterList.h"
26
27 #include "KokkosClassic_config.h"
28 #include "CKokkos_DefaultNode.h"
29
30 #if   defined(HAVE_KOKKOSCLASSIC_THREADPOOL)
31 #include "CKokkos_TPINode.h"
32 #endif
33
34 #include "CTpetra_DefaultPlatform.h"
35
36 #include "CTpetra_MpiPlatform.h"
37 #include "CTpetra_Map.h"
38 #include "CTpetra_MultiVector.h"
39 #include "CTpetra_CrsMatrix.h"
40
41
42 int main(int argc, char *argv[])
43 {
44 #ifdef HAVE_CTRILINOS_KOKKOS
45 #ifdef HAVE_KOKKOSCLASSIC_THREADPOOL
46
47
48   CT_LocalGlobal_E_t lg = CT_LocalGlobal_E_GloballyDistributed;
49
50   CT_Teuchos_Comm_I_ID_t CommID;
51   CT_Kokkos_TPINode_ID_t NodeID;
52   CT_Tpetra_MpiPlatform_KTPI_ID_t id;
53
54   CT_Tpetra_Map_I_I_KTPI_ID_t Map;
55   CT_Tpetra_MultiVector_D_I_I_KTPI_ID_t x,y;
56   CT_Tpetra_CrsMatrix_D_I_I_KTPI_KDS_ID_t A;
57
58   CT_Teuchos_ParameterList_ID_t paramsID;
59
60   CT_Teuchos_ArrayRCP_D_ID_t yout;
61   CT_Teuchos_ArrayView_cI_ID_t myArrayView;
62   CT_Teuchos_ArrayView_cD_ID_t DcViewID;
63   CT_Teuchos_ArrayView_cI_ID_t cViewID;
64   CT_Teuchos_Array_I_ID_t AID;
65   CT_Teuchos_Array_D_ID_t DAID;
66
67   CT_Teuchos_Tuple_I2_ID_t Tuple_I2;
68   CT_Teuchos_Tuple_I3_ID_t Tuple_I3;
69   CT_Teuchos_Tuple_D2_ID_t Tuple_D2;
70   CT_Teuchos_Tuple_D3_ID_t Tuple_D3;
71
72   int IndexBase = 0;
73   boolean zeroOut = TRUE;
74   size_t i;
75   int j;
76   double value;
77   size_t NumGlobalElements = 40;
```

```
78    size_t numMyElements;
79    const int *MyGlobalElements;
80    double *yy;
81
82    MPI_Init(&argc, &argv);
83    id = Tpetra_DefaultPlatform_getDefaultPlatform();
84    CommID = Tpetra_MpiPlatform_KTPI_getComm(id);
85    NodeID = Tpetra_MpiPlatform_KTPI_getNode(id);
86
87    /* Creating Map and extracting the numbering of its elements */
88    Map = Tpetra_Map_I_I_KTPI_Create(NumGlobalElements, IndexBase, CommID,
          lg, NodeID);
89    numMyElements = Tpetra_Map_I_I_KTPI_getNodeNumElements(Map);
90    myArrayView = Tpetra_Map_I_I_KTPI_getNodeElementList(Map);
91    MyGlobalElements = (int*)malloc(numMyElements*sizeof(int));
92    if (MyGlobalElements == NULL) {
93      fprintf(stderr,"Couldn't␣malloc␣for␣MyGlobalElements\n");
94      printf( "\nEnd␣Result:␣TEST␣FAILED\n" );
95      return 1;
96    }
97    MyGlobalElements = Teuchos_ArrayView_cI_getRawPtr(myArrayView);
98
99    /* Creating and filling a Vector */
100   x = Tpetra_MultiVector_D_I_I_KTPI_Create(Map,1,zeroOut);
101   for (i=0; i<numMyElements; i++) {
102     j = MyGlobalElements[i];
103     value=1.0*(double)j;
104     Tpetra_MultiVector_D_I_I_KTPI_replaceGlobalValue(x,j,0,value);
105   }
106
107   Tuple_I2 = Teuchos_Tuple_I2_Create();
108   Tuple_I3 = Teuchos_Tuple_I3_Create();
109   Tuple_D2 = Teuchos_Tuple_D2_Create();
110   Tuple_D3 = Teuchos_Tuple_D3_Create();
111
112   AID = Teuchos_Array_I_Create();
113   DAID = Teuchos_Array_D_Create();
114
115   /* Creating and filling a sparse matrix */
116   paramsID = Teuchos_ParameterList_Create();
117   A = Tpetra_CrsMatrix_D_I_I_KTPI_KDS_Create_AllRows
118         (Map,3,CT_ProfileType_E_DynamicProfile,paramsID);
119
120 for (i=0; i<numMyElements; i++) {
121
122     if (MyGlobalElements[i] == 0) {
123      Tuple_I2 = Teuchos_Tuple_I2_tuple(MyGlobalElements[i],
           MyGlobalElements[i]+1);
124      AID = Teuchos_Array_I_New_FromTuple_2(Tuple_I2);
125      cViewID=Teuchos_ArrayView_I_getConst(Teuchos_Array_I_Iview(AID,0,2));
126
127      Tuple_D2 = Teuchos_Tuple_D2_tuple(2.0,-1.0);
128      DAID = Teuchos_Array_D_New_FromTuple_2(Tuple_D2);
129      DcViewID=Teuchos_ArrayView_D_getConst(Teuchos_Array_D_Iview(DAID,0,2)
```

```
        );
130
131     Tpetra_CrsMatrix_D_I_I_KTPI_KDS_insertGlobalValues
132         (A,MyGlobalElements[i],cViewID,DcViewID);
133    }
134
135    else if (MyGlobalElements[i] == NumGlobalElements-1) {
136     Tuple_I2 = Teuchos_Tuple_I2_tuple(MyGlobalElements[i]-1,
           MyGlobalElements[i]);
137     AID = Teuchos_Array_I_New_FromTuple_2(Tuple_I2);
138     cViewID=Teuchos_ArrayView_I_getConst(Teuchos_Array_I_Iview(AID,0,2));
139
140     Tuple_D2 = Teuchos_Tuple_D2_tuple(-1.0,2.0);
141     DAID = Teuchos_Array_D_New_FromTuple_2(Tuple_D2);
142     DcViewID=Teuchos_ArrayView_D_getConst(Teuchos_Array_D_Iview(DAID,0,2)
           );
143
144     Tpetra_CrsMatrix_D_I_I_KTPI_KDS_insertGlobalValues
145         (A,MyGlobalElements[i],cViewID,DcViewID);
146    }
147
148    else {
149     Tuple_I3 = Teuchos_Tuple_I3_tuple(MyGlobalElements[i]-1,
           MyGlobalElements[i],
150                                       MyGlobalElements[i]+1);
151     AID = Teuchos_Array_I_New_FromTuple_3(Tuple_I3);
152     cViewID=Teuchos_ArrayView_I_getConst(Teuchos_Array_I_Iview(AID,0,3));
153
154     Tuple_D3 = Teuchos_Tuple_D3_tuple(-1.0,2.0,-1.0);
155     DAID = Teuchos_Array_D_New_FromTuple_3(Tuple_D3);
156     DcViewID=Teuchos_ArrayView_D_getConst(Teuchos_Array_D_Iview(DAID,0,3)
           );
157
158     Tpetra_CrsMatrix_D_I_I_KTPI_KDS_insertGlobalValues
159         (A,MyGlobalElements[i],cViewID,DcViewID);
160    }
161
162 }
163
164 Tpetra_CrsMatrix_D_I_I_KTPI_KDS_fillComplete(A,paramsID);
165
166 y = Tpetra_MultiVector_D_I_I_KTPI_Create(Map,1,zeroOut);
167 Tpetra_CrsMatrix_D_I_I_KTPI_KDS_apply(A,x,y,CT_ETransp_E_NO_TRANS
       ,1.0,0.0);
168
169 yout = Tpetra_MultiVector_D_I_I_KTPI_getDataNonConst(y,0);
170 yy=Teuchos_ArrayRCP_D_get(yout);
171
172 for (i=0; i<numMyElements; i++) {
173   printf("%f\n",yy[i]);
174 }
175
176  Teuchos_Array_I_Destroy(&AID);
177  Teuchos_Array_D_Destroy(&DAID);
```

```
178   Teuchos_ArrayView_cI_Destroy(&cViewID);
179   Teuchos_ArrayView_cD_Destroy(&DcViewID);
180   Teuchos_ArrayRCP_D_Destroy(&yout);
181   Tpetra_CrsMatrix_D_I_I_KTPI_KDS_Destroy(&A);
182   Tpetra_MultiVector_D_I_I_KTPI_Destroy(&x);
183   Tpetra_MultiVector_D_I_I_KTPI_Destroy(&y);
184   Tpetra_Map_I_I_KTPI_Destroy(&Map);
185   Tpetra_MpiPlatform_KTPI_Destroy(&id);
186   Kokkos_TPINode_Destroy(&NodeID);
187
188 #endif /* HAVE_KOKKOSCLASSIC_THREADPOOL */
189 #endif /* HAVE_CTRILINOS_KOKKOS */
190 }
```

A.3. ForTrilinos Procedural Fortran Sample Application

The source code shown in listing A.3 illustrates the use of Fortran procedural bindings by an external application. The interfaces used in this code correspond to the first ForTrilinos sublayer shown in figure 2.1. In this layer the procedural interfaces correlate with the headers in the CTrilinos layer. Since this layer makes no use of object-oriented programming features the end application seems as complex as the previously shown implementation in listing A.2.

The explicit use of procedural bindings also makes this layer counterintuitive to Fortran programmers as we are forced to use struct IDs for Teuchos array classes instead of allocatable arrays which are native Fortran features. Allocatable arrays defer to the compiler memory management associate with the use of an array, memory is dynamically allocated when the array is created and is automatically deallocated by the compiler when the variable goes out of scope.

In this implementation, similar to the CTrilinos implementation, the procedures and struct IDs used to wrap the underlying C++ methods and classes respectively are identified by a label for the instances of the templated parameters used. This approach allows the compiler to differentiate between the different implementation but at the same time makes the application source code less flexible. In contrast to the C++ implementation were the hardware considerations can be changed by modifying only one line in the code (see line 27 in listing A.1), the CTrilinos and ForTrilinos procedural bindings implementation (in listings A.2 and A.3 respectively) required the modification the labels use throughout the application.

The interfaces of both the CTrilinos layer and the ForTrilinos procedural bindings are not intended for the end user, they exist only to guarantee the portability of the software stack as they support the ultimate object-oriented Fortran interfaces that are part of the ForTrilinos package. The procedural layer in CTrilinos and ForTrilinos hide the underlying complexity of a software stack that is both platform and compiler independent and enable OO Fortran interfaces that have idioms that should feel natural to Fortran programmers.

Listing A.3: Source code for sample Fortran procedural application using ForTrilinos Fortran procedural bindings in the ForTrilinos library package.

```
#include "ForTrilinos_config.h"
program main
#ifdef HAVE_KOKKOSCLASSIC_THREADPOOL
  use mpi
  use iso_c_binding ,only : c_int,c_double,c_bool,c_f_pointer
  use iso_fortran_env ,only : error_unit ,output_unit
  use fortrilinos_utils ,only : valid_kind_parameters
  use fortpetra
  use forteuchos

  implicit none

  !
  ! Data declarations
  !

  integer(c_size_t) :: NumGlobalElements = 40_c_size_t
  integer(c_size_t) numMyElements
  type(c_ptr) :: MyGlobalElements_ptr
  integer(c_int), pointer :: MyGlobalElements(:) => NULL()
  type(FT_Teuchos_Comm_ID_t) commID
  type(FT_Kokkos_TPINode_ID_t) NodeID
  type(FT_Tpetra_MpiPlatform_ID_t) id

  type(FT_Teuchos_ArrayView_ID_t) myArrayView

  integer(c_int) :: IndexBase = 0_c_int
  type(FT_Tpetra_Map_ID_t) Map

  type(FT_Tpetra_MultiVector_ID_t) x,y

  type(FT_Tpetra_CrsMatrix_ID_t) A

  type(FT_Teuchos_ParameterList_ID_t) paramsID
  type(FT_Teuchos_ArrayView_ID_t) cViewID
  type(FT_Teuchos_ArrayView_ID_t) DcViewID

  type(FT_Teuchos_Array_ID_t) AID
  type(FT_Teuchos_Array_ID_t) DAID

  type(FT_Teuchos_Tuple_ID_t) Tuple_I2
  type(FT_Teuchos_Tuple_ID_t) Tuple_I3
  type(FT_Teuchos_Tuple_ID_t) Tuple_D2
  type(FT_Teuchos_Tuple_ID_t) Tuple_D3

  integer(c_size_t) i
  integer(c_int) j
  real(c_double) value

  type(FT_Teuchos_ArrayRCP_ID_t) yout
  type(c_ptr) :: yy_ptr
```

```
52   real(c_double), pointer :: yy(:) => NULL()
53
54   integer :: ierr
55   !
56   ! Executable code
57   !
58
59   call MPI_INIT(ierr)
60   id = Tpetra_DefaultPlatform_getDefaultPlatform()
61   CommID = Tpetra_MpiPlatform_KTPI_getComm(id)
62   NodeID = Tpetra_MpiPlatform_KTPI_getNode(id)
63
64   !Creating Map and extracting the numbering of its elements
65   Map = Tpetra_Map_I_I_KTPI_Create(NumGlobalElements, IndexBase, CommID, &
66                                    FT_LocalGlobal_E_GloballyDistributed,
                                        NodeID)
67   numMyElements = Tpetra_Map_I_I_KTPI_getNodeNumElements(Map)
68   myArrayView = Tpetra_Map_I_I_KTPI_getNodeElementList(Map)
69
70   MyGlobalElements_ptr = Teuchos_ArrayView_cI_getRawPtr(myArrayView)
71   call c_f_pointer(MyGlobalElements_ptr, MyGlobalElements, [numMyElements
        ])
72
73   x = Tpetra_MultiVector_D_I_I_KTPI_Create(Map,1_c_size_t,FT_TRUE)
74   do i=1,numMyElements
75     j = MyGlobalElements(i)
76     value = dble(j)
77    call Tpetra_MultiVector_D_I_I_KTPI_replaceGlobalValue(x,j,0_c_size_t,
         value)
78   enddo
79
80   Tuple_I2 = Teuchos_Tuple_I2_Create()
81   Tuple_I3 = Teuchos_Tuple_I3_Create()
82   Tuple_D2 = Teuchos_Tuple_D2_Create()
83   Tuple_D3 = Teuchos_Tuple_D3_Create()
84
85   AID = Teuchos_Array_I_Create()
86   DAID = Teuchos_Array_D_Create()
87
88   !Creating and filling a sparse matrix
89   paramsID = Teuchos_ParameterList_Create()
90   A = Tpetra_CrsMatrix_D_I_I_KTPI_KDS_Create_AllRows(Map,3_c_size_t, &
91                    FT_ProfileType_E_DynamicProfile,paramsID)
92
93   do i=1, numMyElements
94
95     if (MyGlobalElements(i) == 0_c_int) then
96       Tuple_I2 = Teuchos_Tuple_I2_tuple(MyGlobalElements(i),
            MyGlobalElements(i)+1_c_int)
97       AID = Teuchos_Array_I_New_FromTuple_2(Tuple_I2)
98       cViewID=Teuchos_ArrayView_I_getConst(Teuchos_Array_I_Iview(AID,0
            _c_int,2_c_int))
99
100      Tuple_D2 = Teuchos_Tuple_D2_tuple(2.0_c_double,-1.0_c_double)
```

```
101       DAID = Teuchos_Array_D_New_FromTuple_2(Tuple_D2)
102       DcViewID=Teuchos_ArrayView_D_getConst(Teuchos_Array_D_Iview(DAID,0
              _c_int,2_c_int))
103
104       call Tpetra_CrsMatrix_D_I_I_KTPI_KDS_insertGlobalValues(A,
              MyGlobalElements(i), &
105                       cViewID,DcViewID)
106
107     elseif (MyGlobalElements(i) == NumGlobalElements-1_c_int) then
108       Tuple_I2 = Teuchos_Tuple_I2_tuple(MyGlobalElements(i)-1_c_int,
              MyGlobalElements(i))
109       AID = Teuchos_Array_I_New_FromTuple_2(Tuple_I2)
110       cViewID=Teuchos_ArrayView_I_getConst(Teuchos_Array_I_Iview(AID,0
              _c_int,2_c_int))
111
112       Tuple_D2 = Teuchos_Tuple_D2_tuple(-1.0_c_double,2.0_c_double)
113       DAID = Teuchos_Array_D_New_FromTuple_2(Tuple_D2)
114       DcViewID=Teuchos_ArrayView_D_getConst(Teuchos_Array_D_Iview(DAID,0
              _c_int,2_c_int))
115
116       call Tpetra_CrsMatrix_D_I_I_KTPI_KDS_insertGlobalValues(A,
              MyGlobalElements(i), &
117                       cViewID,DcViewID)
118
119     else
120       Tuple_I3 = Teuchos_Tuple_I3_tuple(MyGlobalElements(i)-1_c_int,
              MyGlobalElements(i),&
121                                         MyGlobalElements(i)+1_c_int)
122       AID = Teuchos_Array_I_New_FromTuple_3(Tuple_I3)
123       cViewID=Teuchos_ArrayView_I_getConst(Teuchos_Array_I_Iview(AID,0
              _c_int,3_c_int))
124
125       Tuple_D3 = Teuchos_Tuple_D3_tuple(-1.0_c_double,2.0_c_double,-1.0
              _c_double)
126       DAID = Teuchos_Array_D_New_FromTuple_3(Tuple_D3)
127       DcViewID=Teuchos_ArrayView_D_getConst(Teuchos_Array_D_Iview(DAID,0
              _c_int,3_c_int))
128
129       call Tpetra_CrsMatrix_D_I_I_KTPI_KDS_insertGlobalValues(A,
              MyGlobalElements(i), &
130                       cViewID,DcViewID)
131
132     endif
133
134   enddo
135
136   call Tpetra_CrsMatrix_D_I_I_KTPI_KDS_fillComplete(A,paramsID)
137
138   y = Tpetra_MultiVector_D_I_I_KTPI_Create(Map,1_c_size_t,FT_TRUE)
139   call Tpetra_CrsMatrix_D_I_I_KTPI_KDS_apply(A,x,y,FT_ETransp_E_NO_TRANS,
          &
140                                              1.0_c_double,0.0_c_double)
141
142   yout = Tpetra_MultiVector_D_I_I_KTPI_getDataNonConst(y,0_c_size_t)
```

```
143   yy_ptr=Teuchos_ArrayRCP_D_get(yout)
144   call c_f_pointer (yy_ptr, yy, [numMyElements])
145
146   do i=1, numMyElements
147     print *, yy(i)
148   enddo
149 #endif /* HAVE_KOKKOSCLASSIC_THREADPOOL */
150 end program
```

A.4. ForTrilinos Object-Oriented Fortran Sample Application

The sample implementation for the end user OO Fortran application is shown in listing A.5. This corresponds to the top layer of the software stack discussed in section 2.3. In this implementation, objects are created through class named, overloaded user-defined constructors, making the interface less complex. As was the case in the C++ implementation, here we are able to modified hardware configuration for the application by changing lines 12 and 13. The data type of the templated parameters are declared in module `my_parameters`. It is done in a separate module by design, to isolate the source code which can undergo possible modification. This module correspond to the typedefs shown in listing A.1.

The specific instances of the PDTs that correspond to the underlying instances of the Trilinos template classes are declared in lines 29-33. A clean syntax is achieved in this layer by defining type-bound procedures that are invoked through a unified generic interface (see lines 89 and 92). The interface is used to overload the implementations associated with a specific instantiation of the PDT that wraps the corresponding instance of the underlying C++ template class.

In this implementation, variables of intrinsic data types are defined using the `kind` parameter that corresponds to the templated parameters used in the application. The variable declaration shown in line 35 could also be expressed using the syntax shown in listing A.4. In both cases, the variable `MyGlobalElements` has the same kind parameter, however, the use of `ft_selected` (defined in module `ForTrilinos_PDT_enums.F90` not shown) unifies the definitions of all `kind` parameters and encapsulate in `my_parameters` module the required source code modifications to account for data type changes. The approach followed within the implementation promotes consistency, avoids interoperability issues, and insures compile-time error checking.

Listing A.4: Alternative syntax for intrinsic data type declaration in OO ForTrilinos enable application

```
integer(kind=c_int64_t), allocatable , dimension(:) :: MyGlobalElements
```

A utility module (`ForTrilinos_PDT_utils.F90` not shown) was developed within ForTrilinos to encapsulate special functionality required to simplified the interfaces with which the end user interacts. The functionality includes the conversion of Teuchos::Array, Teuchos::ArrayView and Teuchos::ArrayRCP to allocatable Fortran arrays, which are what Fortran users would expect

to use as arguments to the different type-bound procedures. The module also includes functionality for the inverse conversion to be able to pass ForTrilinos procedural bindings the data in the expected format. The capabilities within this module address one of the main requirements of this project, the development of an OO interface that feels natural to Fortran programmers.

Listing A.5: Source code for sample object-oriented Fortran application using ForTrilinos library package.

```
1  module my_parameters
2    use ForTrilinos_PDT_enums, only : ft_int_e, ft_long_e, ft_double_e, &
3                                      ft_size_t_e, ft_TPINode_e, ft_KDS_e, &
4                                      ft_selected
5    use FKokkos_TPINode, only: TPINode
6    integer, parameter :: Ordinal=ft_int_e
7    integer, parameter :: Scalar=ft_double_e
8    integer, parameter :: LocalOrdinal=ft_int_e
9    integer, parameter :: GlobalOrdinal=ft_long_e
10   integer, parameter :: SizeType=ft_size_t_e
11   integer, parameter :: LocalMatOps=ft_KDS_e
12   integer, parameter :: Node=ft_TPINode_e
13   type(TPINode) :: MyNode
14 end module
15 program main
16   use my_parameters
17   use FTpetra_Map, only : Map
18   use FTpetra_MultiVector, only : MultiVector
19   use FTpetra_CrsMatrix, only : CrsMatrix
20   use FTpetra_DefaultPlatform
21   use FTpetra_MpiPlatform, only : MpiPlatform
22   use FTeuchos_ParameterList, only : Teuchos_ParameterList
23   use FTeuchos_Comm, only : Comm
24   use ForTrilinos_enum_wrappers, only :
       FT_LocalGlobal_E_GloballyDistributed, &
25     FT_ProfileType_E_DynamicProfile,FT_ETransp_E_NO_TRANS
26   implicit none
27
28   ! Specify types use in this example
29   type(Map(LocalOrdinal,GlobalOrdinal,Node)) :: map
30   type(MultiVector(Scalar,LocalOrdinal,GlobalOrdinal,Node)) :: x, y
31   type(CrsMatrix(Scalar,LocalOrdinal,GlobalOrdinal,Node,LocalMatOps)) :: A
32   class(Comm(Ordinal)), allocatable :: MyComm
33   type(MpiPlatform(Node)) :: Platform
34   type(ParameterList) :: param
35   integer(kind=ft_selected(GlobalOrdinal)), allocatable, dimension(:) ::
       MyGlobalElements
36   integer(kind=ft_selected(LocalOrdinal)), allocatable, dimension(3) ::
       Indices
37   real(kind=ft_selected(Scalar)), allocatable, dimension(3) :: Val
38   real(kind=ft_selected(Scalar)), allocatable, dimension(:) ::
       y_local_vals
39   integer(kind=ft_selected(SizeType)) :: NumMyElements, j, i
40
41   ! Parameters
42   real(kind=ft_selected(Scalar)) :: alpha=1.0, beta=0.0, real_one=1.0,
```

```
      real_two=2.0
43  integer(kind=ft_selected(SizeType)) :: NumGlobalElements=10, VecIndex=1,
        NumEntries=3
44  integer(kind=ft_selected(LocalOrdinal)) :: one=1, IndexBase=1
45
46  ! Get communicator
47  Platform = getDefaultPlatform()
48  MyComm = Platform%getComm()
49  MyNode = Platform%getNode()
50
51  !Creating Map and extracting the numbering of its elements
52  map = Map(NumGlobalElements, IndexBase, MyComm, &
53            FT_LocalGlobal_E_GloballyDistributed, MyNode)
54  NumMyElements = Map%getNodeNumElements()
55  MyGlobalElements = Map%getNodeElementList()
56
57  ! Create MultiVectors
58  x = MultiVector(map,VecIndex,FT_TRUE)
59  y = MultiVector(map,VecIndex,FT_TRUE)
60  do i=1,numMyElements
61   call x%replaceGlobalValue(MyGlobalElements(i),VecIndex,(real_one*
        MyGlobalElements(i))
62  enddo
63
64  ! Create a CrsMatrix using the map, with a dynamic allocation of 3
       entries per row
65  param = ParameterList()
66  A = CrsMatrix(map,NumEntries,FT_ProfileType_E_DynamicProfile,param)
67
68  Val(1) = -real_one
69  Val(2) =  real_two
70  Val(3) = -real_one
71  do i=1,NumMyElements
72    if (MyGlobalElements(i)==one) then
73      Indices(1) = MyGlobalElements(i)
74      Indices(2) = MyGlobalElements(i) + one
75      call A%insertGlobalValues(MyGlobalElements(i),Indices(1:2),Val(1:2))
76    elseif (MyGlobalElements(i)==NumGlobalElements) then
77      Indices(1) = MyGlobalElements(i) - one
78      Indices(2) = MyGlobalElements(i)
79      call A%insertGlobalValues(MyGlobalElements(i),Indices(1:2),Val(1:2))
80    else
81      Indices(1) = MyGlobalElements(i) - one
82      Indices(2) = MyGlobalElements(i)
83      Indices(3) = MyGlobalElements(i) + one
84      call A%insertGlobalValues(MyGlobalElements(i),Indices(1:3),Val(1:3))
85    endif
86  enddo
87
88  ! Complete the fill, ask that storage be reallocated and optimized
89  call A%fillComplete(param)
90
91  ! Matrix-Vector Multiply
92  call A%apply(x,y,FT_ETransp_E_NO_TRANS,alpha,beta)
```

```
93
94    ! Output Y
95    y_local_vals = y%getDataNonConst(VecIndex)
96    do i=1,NumMyElements
97      print *, y_local_vals(i)
98    enddo
99  end
```

DISTRIBUTION:

1	MS 0899	Technical Library, 9536 (electronic copy)
1	MS 0359	D. Chavez, LDRD Office, 1911

Sandia National Laboratories